Building Shared Libraries with VB6

Jim White
mathimagics@yahoo.co.uk
Canberra, Australia
July 2006

Abstract

VB6 programs at runtime make significant use of COM interfaces, even if the source code does not use COM objects explicitly. Some critical runtime support mechanisms, including error handling, are COM-based (eg. the **Err** object). VB6 applications (EXE's) automatically initialise their COM environment when executed, but VB6 DLL's only initialise COM when the client application uses the DLL via a COM interface, not as an API (a shared library).

If a client application calls a VB6 DLL function directly, the DLL's runtime state has no COM capability, and so the DLL will abort if it makes any sort of COM object reference. This has fatal consequences for the entire client process.

We present a method for building a VB6 DLL that can initialise its COM environment automatically. This allows VB6 DLL's to be deployed as shared libraries, so they can be serve as API's for clients written in other languages.

We begin by showing how a client can manually initialise a VB6 library, via the DLL's ***DllGetClassObject*** function. Then we show how this method can be automated by implementing it in the DLL's own *entrypoint* function.

Finally, we look at the issue of VB6 **Form** display, and show that a VB6 library can operate a non-modal GUI, even with a non-VB client, with some careful programming.

1. Introduction

It is a well-known fact that VB6 can create DLL's – it does just that when compiling and linking an ***ActiveX DLL*** project. These DLL's, however, are constructed with a specific purpose in mind – they allow a VB6 program to be a COM object provider. In OO terminology, an ActiveX DLL is intended to allow VB6 to produce *public class libraries*.

Public class libraries are managed by the Operating System's OLE/COM layer. When a client application requests an instance of apublic class (the VB6 equivalent of a ***CreateObject*** call), the OS locates and loads the appropriate server DLL (the *object provider*), and calls the DLL's COM interface functions to handle the client request.

That is why all VB6 DLL's export the same set of functions (***DllGetClassObject,*** etc). These provide a common interface for DLL's that support the OLE/COM model. This interface is documented in the MSDN library.

2. ActiveX DLL Structure

If you examine the *export table* of an ActiveX DLL, you will always see the same list of exported functions:

- *DllGetClassObject*
- *DllCanUnloadNow*
- *DllRegisterServer (*)*
- *DllUnregisterServer (*)*

These functions form the standard interface defined for DLLs that support the OLE Component Object Model (COM). The last two, those indicated with (*), are in fact not used at all at runtime by clients, they are only used when registering or de-registering the DLL (the process by which a class library is made public).

DllCanUnloadNow is used to notify the DLL when a client releases (destroys) its last reference to any object created by it.

DllGetClassObject is the principal interface routine – this is the function that OLE calls when the client executes a ***CreateObject*** request.

3. VB6 and Private Class Libraries

VB6 is very much based on COM technology. Even if you don't use classes in your code, every program you write in VB6 makes extensive use of the COM object model. Forms and controls are COM objects, as are private classes, and various *global* objects such as **Err** and **App**.

These objects are not public, however. Every VB6 module (EXE or DLL) has a built-in *private class library,* which provides the COM interface for these VB-specific objects.

This explains why VB6's built-in controls can't be created by a non-VB application – there is no public class library that provides them.

4. ActiveX DLL's as Function Libraries

Building a VB6 DLL that can be used as function library is quite simple. This is simply a matter of getting selected functions to be included in the DLL's *export table.* Exporting functions from ACtiveX DLL's is not a supported feature, but is easily achieved with a VB6 *link control tool.*

VB6 has its own compiler and linker utilities, which are invoked when a project is compiled to native code. The VB6 compile-and-link mechanism was first described by Lee Thé in 1997 (**[**2**]**). Methods of customising this process followed soon after, notably John Chamberlain's article *"Taking Control of the Compile Process"* in 1999 **[**1**].** A simpler method based on command-line interception was described by P.J. Morris in **[**3**].**

For our purpose, we have no need to modify the compilation step, we only need to customise the final linkage. This can be done with a simple *VB6 Link Control* tool – the source code for a suitable tool is provided in the appendices to this document.

The link customisations required to build the VB6 library examples described below are elementary - we need only a facility to nominate functions for exporting, and the ability to modify the DLL's entrypoint address.

If you are unfamiliar with VB6 compile/link customisation methods, the link tool we provide here presents the easiest way to implement the examples. Morris's article **[**3**]** is well worth reading in order to understand the principles involved.

5. Building a New VB6 Function Library

The general method for creating any new VB6 library DLL is as follows:

- Use an ActiveX DLL project template
- Create a dummy class (there is no need to write any code for it), and set its **Instancing** property to 5 (Multiuse). This ensures the project will produce a DLL when the MAKE option is invoked.
- Compile the empty "skeleton" project to produce an initial, dummy version of the DLL
- Now go to the Project Properties menu, access the "Compatibility" tab, and set the project to "Binary Compatibility", nominating the DLL you just made. This prevents VB6 from producing new (and useless) GUID's every time that you recompile the DLL
- Put the required library functions (those that are to be exported) in one or more BAS modules
- Create a link control command file to specify the exported functions

Appendix I of this document contains the complete source code and instructions for a simple but effective link-control tool called MVBLC (Mathimagics VB Link Controller). With this tool you can control the DLL linking (exporting functions, etc) by creating simple link command files (VBC files). When you MAKE any project in VB6, it will be compiled and linked normally unless it is a DLL and there is a VBC file present.

If you already have a similar tool, you should be able to use that – the only link customisations that we need to perform are the specification of exported functions and the option to change the DLL's *entrypoint* address. If you wish to use MVBLC, then now is a good time to install it (see Appendix I).

6. Initialisation of VB6 Function Libraries

Building a VB6 library DLL is simple, but getting it to provide the desired functionality is much less straight-forward. Even without introducing complexities such as **Forms**, what can appear to be "elementary" VB6 code is often found to crash when the DLL is called by a client application.

As we discussed earlier, the VB6 runtime environment is very much dependent on COM, even a project with no Forms, just "pure" module functions, can be quite dependent on COM mechanisms to function correctly.

All VB6 executables (whether EXE's or DLL's) perform two types of initialisation:

- Generic - initialisation of the VB6 runtime environment, including initialisation of the runtime heap structures and the exception handling mechanism
- COM - the private class library embedded in all VB6 executables (this includes not just Forms, if present, but also other "global" COM objects such as **Err**)

Without the COM initialisation, a VB6 DLL cannot execute any COM-related statement. It cannot access the **Err** object, for example, nor can it load a **Form** or create an instance of a private **Class**. Another mechanism that is dependent on COM initialisation is the way VB6 performs CALL's from an application to external functions defined with the VB6 **Declare** statement – the way VB6 does this at runtime also involves the **Err** object. So we can't even make API calls (at least, not this way) without COM initialisation.

7. ActiveX DLL Initialisation

How are ActiveX DLL's normally initialised? Since the ActiveX DLL project was designed as way of building COM object providers, all COM environment initialisation for a VB6 DLL is embedded in the DLL's COM interface routines. This code gets executed automatically when a client application makes the first ***CreateObject*** request on a class for which this DLL is the registered server.

These requests are handled by the OLE32 API, in particular the ***CoGetClassObject*** function. Once the DLL has been identified via the registry, it is loaded via ***CoLoadLibary*** (the OLE equivalent of ***LoadLibary***).

ActiveX DLL's are like any other DLL in one respect - loading the DLL automatically executes the DLL's entrypoint function, usually called ***DllMain*** in non-VB DLL's, but in a VB6 DLL its name is ***__vbaS***.

Note that the name ***DllMain*** is typically used to refer to a DLL's *entrypoint* - this entrypoint specifies the address of a function the the OS will call when the DLL is first loaded into a client process. This DLL entrypoint function is not called by name, but by its address – this address is placed in the DLL's header by the linker when the DLL is built.

Unfortunately, the standard entrypoint function, ***__vbaS,*** that gets generated for an ActiveX DLL *does not include* COM *initialisation*. It simply performs the generic initialisation, such as initialisation of the heap structures (wich are used for dynamic array allocations), and initialisation of the runtime exception handling mechanism.

COM initialisation is only performed when the first ***CreateObject*** request made by from the client application is serviced. The OLE subsystem, after loading the DLL if necessary, then makes a conventional API call (ie. by name) to the DLL's ***DllGetClassObject*** function.

If your library DLL is not intended for general distribution, and you don't mind modifying your client applications individually to make them "compatible" with your library DLL, then a simple way to solve the COM initialisation problem is this:

- register the DLL in the normal COM way
- have your client applications create an instance of your DLL's public class

In other words, the COM initialisation problem can be avoided altogether by making appropriate coding changes to client applications (although this does require the client to be written in a language that supports creation of COM objects).

We would prefer, of course, to solve the problem in a more generic way – one that does not require any specific coding changes in the client (ie. the client should be able to call the VB6 library in just the same way that it calls any other API function). Accordingly, we need to find a non-COM way of initialising

the DLL, and one that can be performed automatically (ie. we want this initialisation to be *transparent* to the client application).

8. COM Initialisation via DllGetClassObject

The key to a general solution to this problem is provided by the standard interface function that all ActiveX DLL's provide, the ***DllGetClassObject*** function. As we saw earlier, ***DllGetClassObject*** is a standard interface, and is described in the MSDN library as follows:

STDAPI DllGetClassObject(

```
  REFCLSID clsid,// CLSID for the class object

  REFIID   riid, // RIID for the interface that communicates
                 // with the class object

  LPVOID  *ppv   // Address of output variable that receives the
                 // interface pointer requested in riid
      );
```

Parameters

```
  rclsid [in]   CLSID that will associate the correct data and code.

  riid   [in]   Reference to the identifier of the interface that
                the caller is to use to communicate with the class
                object. Usually, this is IID_IClassFactory (the
                interface identifier for IClassFactory).

  ppv    [out]  Address of pointer variable that receives the
                interface pointer requested in riid. Upon successful
                return, *ppv contains the requested interface pointer.
                If an error occurs, the interface pointer is NULL.
```

Return Values

```
This function supports the standard return values E_INVALIDARG,
E_OUTOFMEMORY and  E_UNEXPECTED, as well as the following:

  S_OK   (0)    The object was retrieved successfully.

  CLASS_E_CLASSNOTAVAILABLE  (0x80040111)
                The DLL does not support this class.
```

Figure 1 – MSDN entry for COM interface routine **DllGetClassObject**

Note that a ***ClsId*** is just a GUID that uniquely identifies a class from which an object instance is being requested. The DLL instantiates the object, and returns a pointer that identifies both the interface for this class (where the code for the *methods* can be found) and the data corresponding to the this specific *instance* of the class.

When an ActiveX DLL is first referenced by a client, it is in respose to the client's first request to create an object provided by the DLL. As we have seen, the DLL when loaded initially into the client process will only perform generic (non-COM) initialisation.

This implies that the COM initialisation for the DLL must be *automatically* invoked when the client makes the first ***CreateObject*** request. This suggests that one way to COM-initialise a VB6 DLL for library use might be simply to *fabricate a call to its* ***DllGetClassObject*** *function*.

9. COM Initialisation via DllGetClassObject

Note that there are just two input parameters that need to be supplied to ***DllGetClassObject,*** the **clsid** (the GUID for the target class), and the **riid** (the GUID for the COM object-creation interface known as **IClassFactory**). The correct **riid** parameter is system *constant*:

`IID_IClassFactory` = {00000001-0000-0000-C000-000000000046}

We could easily find the correct GUID for **clsid**, too, if we wished – we could register our DLL and then look at its registry settings. In reality, we don't actually need a GUID, nor do we have to register the DLL. If we just pass in a *null pointer* for **clsid** then ***DllGetClassObject*** will return an error code, but it still has to initialise COM, because it can't validate any of the parameters without first *initialisating the COM environment.*

In short, we can get a DLL to perform its COM initialisation by calling its ***DllGetClassObject*** function, passing it an arbitrary (or null) **clsid** value and the constant `IID_IClassFactory.`

Making this ***DllGetClassObject*** call is very easily in the client application, so we will begin with an example in which the client initialises the DLL, then we will investigate automatic DLL self-initialisation.

10. Code Sample 1 - Client-initiated DLL initialisation

This example also provides a simple way of testing the effectiveness of the initialisation method. We show below some simple code that can be pasted into either a VB6 or a PowerBasic client application. It provides a subroutine ***InitVBdll( ),*** which performs the required ***DllGetClassObject*** call in a particular target DLL (here the target is assumed to be *CodeSample1.dll*).

```
Type IID
   data1    As Long
   data2    As Integer
   data3    As Integer
   data4(7) As Byte
   End Type

Declare Function DllGetClassObject Lib "CodeSample1.dll"   _
   Alias "DllGetClassObject"                               _
   (REFCLSID As Long, REFIID As IID, PPV As Long) As Long

Sub InitVBdll()  '  Invoke "COM initialiser" in a VB6 dll
   Dim pIID    As IID
   Dim pDummy  As Long
   '  Set pIID = IID of IClassFactory
   '           = {00000001-0000-0000-C000-000000000046}
   pIID.Data1 = 1
   pIID.Data4(0) = &HC0
   pIID.Data4(7) = &H46
   Call DllGetClassObject(pDummy, pIID, pDummy)
   End Sub
```

Figure 2 – Subroutine **InitVBdll** *- making a "dummy" call to DllGetClassObject*

Next we create a simple VB6 library function for this test – our function is called ***IsPrime***. It accepts a single **Long** parameter, and returns a **Long** result. The return value is 1 if the number passed in is a prime, otherwise the return value is 0. Note that the function does not check for a negative argument, but it does have an error handler which should be invoked if an attempt is made to evaluate **Sqr()** with a negative parameter.

All we need for this example is a standard VB6 ActiveX DLL project, as described above, with a dummy public class, and a module containing our test function. This code is shown below in Figure 3.

```
Option Explicit

Public Function IsPrime(ByVal N As Long) As Long
   Dim i As Long, k As Long
   On Error GoTo BadParameter
   k = Int(Sqr(N))
   For i = 3 To k Step 2
      If (N Mod i) = 0 Then Exit Function
      Next
   IsPrime = 1
   Exit Function

BadParameter:
   MsgBox "Error in IsPrime(): " & Err.Description & Chr$(10) & _
          "         Parameter: &H" & Hex$(N), vbExclamation,    _
          "CodeSample1.dll"
   End Function
```

Figure 3 – VB6 DLL code for CodeSample1 Project

Before compiling the DLL, we need to create the VBC link control file, *CodeSample1.vbc*. This should contain the following commands:

```
Export Module1 IsPrime
Status
```

This assumes we have called the DLL's code module "Module1". The STATUS command is optional, but is recommended because it will always tell you if a custom link was done. That way you can be sure the linker is operating correctly. If the link control system is properly installed, we simply "make" the DLL in the normal way, and if the link is successful, and we have included the STATUS command, the following message should be displayed:

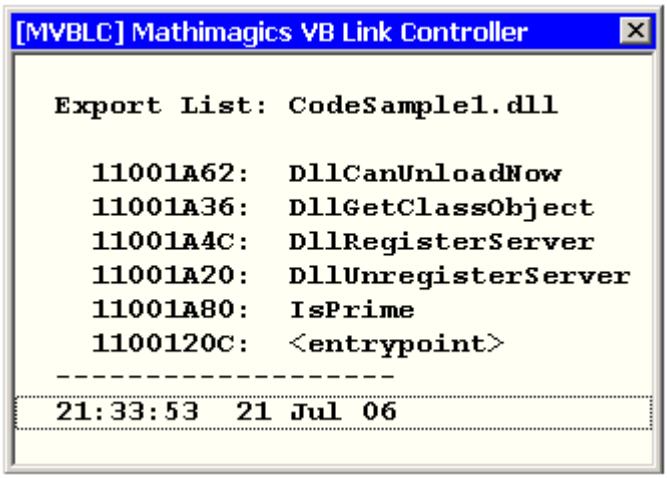

*Figure 4 – **Status** report from the Link Tool*

The export list includes the names that VB6 auto-exports, and also identifies the address of the DLL's *entrypoint* function (a feature we will make use of later). The key entry here is the confirmation that ***IsPrime*** has been exported.

Now we can write a simple client program to test this DLL. This could be in any language (including VB6 of course), but in this document we provide test client programs in "PowerBasic" form.

The code for ***Client1.bas*** is shown below - you just need to paste in the ***InitVBdll*** code from Figure 2 above (the syntax is valid for PowerBasic, so no changes are needed).

```
Declare Function IsPrime Lib "CodeSample1.dll"                _
    Alias "IsPrime"  (ByVal N As Long) As Long

 Sub TestPrime(ByVal N As Long)
    If IsPrime(N) Then
       MsgBox Str$(N) & " is prime", 64, "Client1"
    Else
       MsgBox Str$(N) & " is not prime", 16, "Client1"
       End If
    End Sub

 Function PBMain() As Long

    If MsgBox("Call DLL COM initialiser?",  36,               _
       "Code Sample 1 - Client App") = 6 Then                 _
          Call InitVBdll

    testPrime  41
    testPrime  42
    testPrime -43   ' will trigger DLL error
    End Function
```

Figure 5 – PowerBasic code for ***Client1***

The test client program makes a sequence of 3 calls to ***IsPrime***. If you answer "Yes" to the initial prompt, the DLL should perform correctly and the the following sequence of messages should be seen:

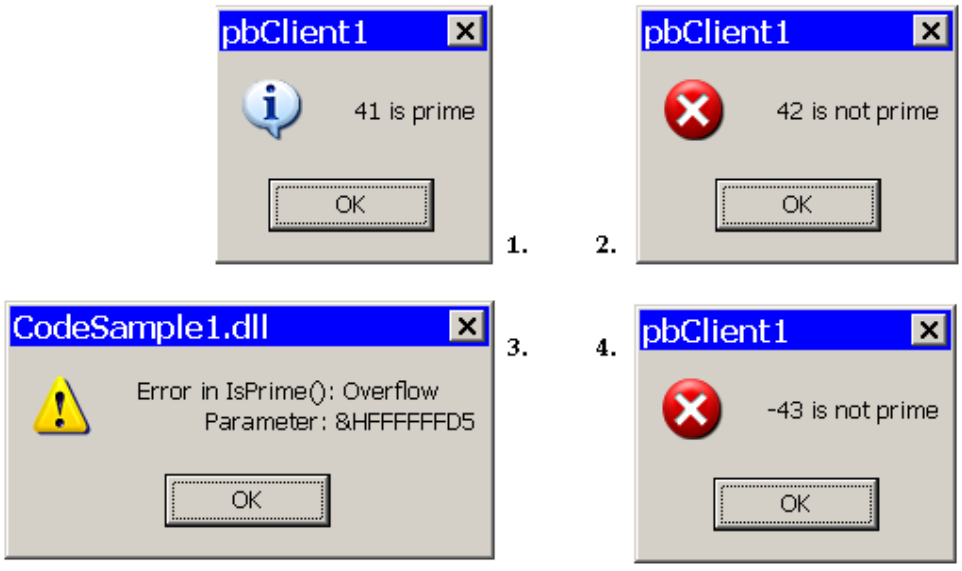

Figure 6 – Messages displayed by **Client1** *when DLL is correctly initialised*

The 3rd message is from the DLL itself, and confirms that the DLL's VB6 runtime error handling mechanism is working.

Responding with "No" at the prompt gives a different result:

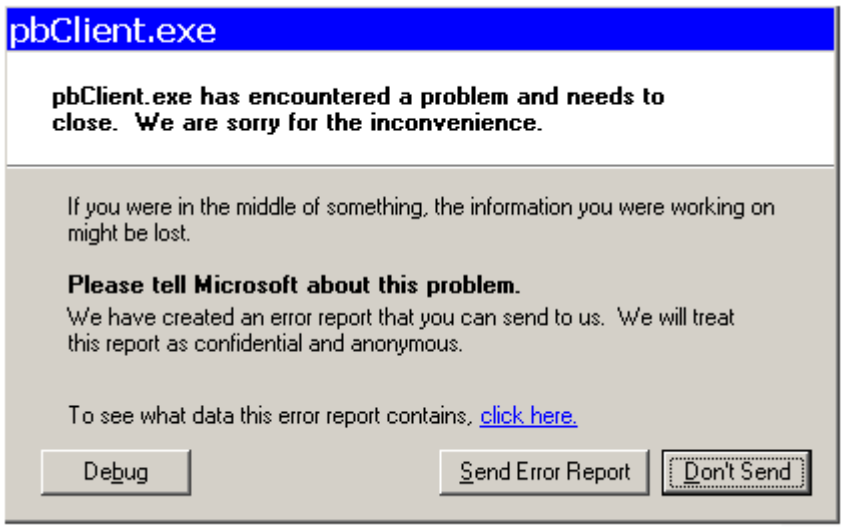

Figure 7 – Client Death Notice (WinXP-style)

Having established that the initialisation method is behaving correctly - we will next consider how we can move the ***InitVBdll*** routine out of the client and into the DLL itself.

11. Automating the DLL initialisation

Every DLL has an *entrypoint* function. This is usually referred to as ***DllMain***, although its name can in fact be anything (at runtime it is called by address, not by name). Entrypoint functions are a standard Windows OS feature – they are called by the OS when the DLL is loaded, unloaded, also whenever the current process begins or ends a thread. One of the parameters for ***DllMain*** is a numeric code that tells the function which particular event has occurred. (For the formal syntax, see the **DllMain** entry in the MSDN library).

This is the general mechanism by which DLL's can perform automatic startup and/or shutdown tasks. The standard entrypoint function provided by VB6 when compiling an ActiveX DLL is called ***__vbaS***. This is the default ***DllMain.*** supplied by VB6 when building ActiveX DLL's. It is actually quite small, serving as a gate to a common function in the VB6 runtime library (MSVBVM60.DLL).

Ideally, we would like the VB6 DLL's we build to automatically invoke our ***InitVBdll*** routine just once, when the entrypoint function is called for the first time. The easiest way to do this is to provide a custom entrypoint function – the entrypoint address is a command-line parameter at link time so a link control tool can easily change the VB6-generated default setting. We can, in fact, provide our own ***DllMain*** from within the VB6 DLL.

A replacement entrypoint function has to behave like a window-message hook. Every call to a DLL's entrypoint function is effectively a message indicating to the DLL that some process-related event has occurred. We still want each message to be processed, so our function should pass all incoming calls on to the real ***__vbaS*** function, ensuring that we do not interfere with the DLL's normal runtime interface. The only action we wish to take ourselves is to call the the COM initialiser when the entrypoint is called for the first time.

The VB6 code for the DLL to perform automatic initialisation is shown below:

```
Option Explicit

Function DllMain(ByVal hInstance As Long, _
                 ByVal lReason   As Long, _
                 ByVal lReserved As Long) As Long

   DllMain = 1    ' this function should always return 1
   Call vbaS(hInstance, lReason, lReserved)
   If lReason = 1 Then Call InitVBdll ' 1 means first call
   End Function

Sub InitVBdll()       ' COM initialiser
   Dim pDummy  As Long
   Dim pIID    As IID ' IID_IClassFactory
                      '  {00000001-0000-0000-C000-000000000046}
   pIID.Data1 = 1
   pIID.Data4(0) = &HC0
   pIID.Data4(7) = &H46
   Call DllGetClassObject(pDummy, pIID, pDummy)
   End Sub
```

Figure 8 – Automatic DLL Initialisation via an Entrypoint Function

This code is very simple, but there remains one problem to solve – how to make the calls to the DLL's own ***DllGetClassObject*** and ***__vbaS*** functions. We can't define them with **Declare** statements, because we can't use the normal VB6 API calling method until initialisation is complete.

An alternative method of making API calls functions without using a **Declare** statement is to declare the functions in a *Type Library*. When a VB6 program calls an API function via a Type Library, the call is made as a normal Win32 API call - via the DLL's *Import Table* – this calling method does not touch the ***Err*** object, so it solves our problem.

So, before we can compile this new DLL code, we first need to construct a small Type Library which will allow the DLL to make calls to its own ***DllGetClassObject*** and ***__vbaS*** functions without prior COM initialisation.

12. About Type Libraries

There are various forms of Type Libraries (TLB's), and different software tools for building them. One format uses a script called **ODL** (Object Definition Language). Although ODL has been superceded by other formats (such as MIDL), it is still a reasonably easy way to generate the simple type library that we need.

One potential disadvantage of this approach is that the target DLL name for each external function declaration is "hardcoded" inside the Type Library, so that is the name that will appear in the DLL's import table. Since we want each different DLL we build to be able to call itself, this would require us to generate a different TLB for each different DLL.

Although creating a TLB is quite simple, making one for each different DLL could easily become tiresome. For each new DLL, we need to make a fresh copy of our ODL script, specifying the target DLL name, and we also have to specify a different GUID for each new Type Library.
Most TLB users do not have this problem - it's quite unusual for a DLL to want to make *external* calls to itself! The typical Type Library used for API function declarations can be used in multiple projects, since the target DLL names are usually constant (eg *kernel32*, *user32*, etc*)*.

Fortunately, there is a reasonably elegant way around this problem - we *can* make a generic "multi-use" TLB that we only need to build once. The idea is simple – we hardcode some generic dummy dll name in our TLB, eg *XXXXXX.dll*. We can then use this TLB with any VB6 DLL project. When we build a DLL, it will contain an Import Table in which the entries ***DllGetClassObject*** and ***vbaS*** are marked as belonging to *XXXXXX.dll*. The Import Table is actually two sub-tables, one which lists the different DLL names, and another which lists each imported function name along with an index into the DLL name table.

So the import DLL names are only stored once, and we can "fix" a DLL after linking by locating its table entry in the DLL file and changing it to the desired value. This is a simple modification that can be performed automatically by the link control tool. The only restriction is that we need to make sure that the TLB's dummy dll name is long enough to allow us to safely overlay it with a different value (we can't change it to name that is longer, otherwise we will corrupt the DLL).

13. Building the Type Library for DLL Self-initialisation

Here is the ODL script we need to create the TLB that will allow us to automatically initialise our VB6 DLL's. The **dllname** entry specifies the "dummy" dll name that we will use (shown in blue). Simply create a text file called *vbLibraryHelper.odl* (the location is arbitrary) and insert the following text:

```
[ uuid(AABBCCDD-0000-0000-0000-000000000000),
  helpstring("MathImagics VB6 DLL Self-initialiser"),
  lcid(0x0), version(1.0)]

  library vbLibraryHelper {
    typedef struct {
      long Data1;
      short Data2;
      short Data3;
      unsigned char Data4[8];} IID;

 [dllname("vbLibraryHelper_mathimagics")]
   module ThisDLL {
    [entry("DllGetClassObject")] Long DllGetClassObject(
          [in] long *pClsId, [in] IID *riid, [in] long *ppv);

    [entry("__vbaS")] Long vbaS(
          [in] long hInst, [in] long lReason, [in] long lRsrvd);
      }
  }
```

Figure 9 – ODL Script for the **vbLibraryHelper** *Type Library*

In ODL script, the **entry** statements correspond to VB6 **Declare** statements, the **uuid** specifies the GUID for the Type Library (arbitrary, but it must be unique), and the "dllname" entry sets the name of the DLL where the declared functions are located. Note that we have to use the name ***vbaS*** in our VB6 code because ***__vbaS*** is regarded as an illegal function name. This ODL associates the two names for us.

We have made the *dummy* dll name fairly long, so we should have no problems overlaying it safely in most DLL's that we are likely to build - if ever a longer setting is needed, you can easily change the TLB, but you will also need to adjust and recompile the MVBLC tool to match the new name).

We make a TLB from an ODL file with the MVS **MkTypLib** tool - this should be present in your Visual Studio directory, if it's missing, you can get it from the MSDN website.

Open a command window in the directory containing the ODL file and execute this command:

mktyplib /nocpp vbLibraryhelper.odl

The response should be a message like this: *Successfully generated type library 'vbLibraryHelper.tlb'*

This TLB file can now be included (ie: *referenced*) in any VB6 project via the *Project → References* option. Then the two function entries will be viewable via the IDE *Object Browser.* The MVBLC link control tool will automatically fix any DLL that uses this TLB.

14. Code Sample 2 – A Self-Initialising VB6 Library

We now have everythig we need to move the initialisation from the client to the DLL. For *CodeSample2*, we simply add the code in Figure 8 into our DLL's main module. We need an extra command in the VBC file to tell the linker to use our ***DllMain*** as the DLL's entrypoint function. Assuming once again that the functions are all located in **Module1**, the VBC commands needed are:

```
Export Module1 IsPrime
Entry  Module1 DllMain
Status
```

The status report for a successful link should look like this:

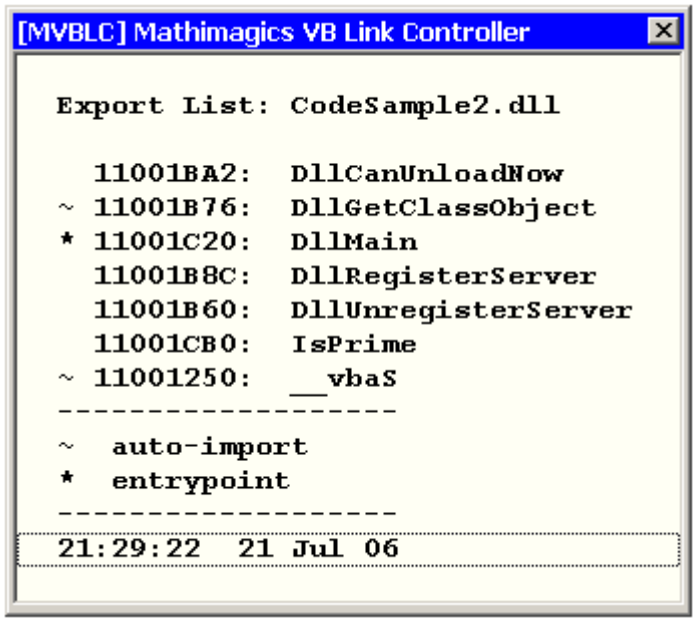

*Figure 10 – **Status** report for CodeSample2*

Notice the differences between the status report above and the earlier one (Figure 4). The export list now includes ***__vbaS*** (this will be exported automatically by MVBLC), and also ***DllMain.*** The latter is also marked as the new entrypoint address. The "Self-Imports" list is a verification that the DLL has been correctly "fixed" to call itself (the TLB dummy name has been successfully replaced).

To test the new DLL, we simply strip ***Client1*** of its DLL initialisation code to produce ***Client2:***

```
Declare Function IsPrime Lib "CodeSample2.dll"                    _
   Alias "IsPrime" (ByVal N As Long) As Long
Sub TestPrime(ByVal N As Long)
   If IsPrime(N) Then
      MsgBox Str$(N) & " is prime", 64, "Client2"
   Else
      MsgBox Str$(N) & " is not prime", 16, "Client2"
      End If
   End Sub
Function PBMain() As Long
   testPrime  41
   testPrime  42
   testPrime -43   ' will trigger DLL error
   End Function
```

Figure 11 – PowerBasic code for ***Client2***

The resulting program should produce exactly the same results as ***Client1***.

15. Forms and Windows

Before we look at introducing **Forms** display to our DLL's, we need to have some idea of how the Windows Messaging System works – a **Form (**and also many standard controls) is both a COM object and an OS object, that is, a *window*. The OS generally doesn't pass window event messages directly to the target windows. Most messages are placed in a queue that the OS keeps for each process.

The processes themselves have to "fetch" these messages from the queues. This means that most GUI applications have one thing in common, a piece of code generally referred to as a ***message pump*****.** When a GUI application starts, it normally starts by creating a main window, and eventually enters "listen" mode by running a message pump, which is just a loop that looks for new messages, and processes them when they appear in the queue.

A VB6 EXE has a built-in message pump, of course. All the *windows-level* interfacing is performed behind the scenes, by the VB6 runtime support library. An ActiveX DLL, however, does NOT have a message pump. It is generally assumed that the ActiveX DLL will be used by a client process that has at least one main window of its own, and is running its own message pump. Furthermore, because VB6 "manages" top-level windows (Forms) as both windows and COM objects, it will not allow a COM server DLL to show non-modal Form's when the client application is not itself a VB6 program.

This is not so much because it can't be done, but it does present some major challenges, such as how to reconcile the clients view of the Form as simply another window with the DLL's view of the Form as a COM object.

This means that an ActiveX DLL can't create windows (apart from MODAL windows, which don't require a pump service) unless the client application is running a pump. If the client application has a GUI, of course, then we don't have a problem.

If the client application is NOT running a pump (eg. it might be a console application, or a C program that doesn't create any windows) then we will need to provide one – this is not an easy thing to do with a DLL, which is expected to return control to the caller, not to sit instead in a message pump loop!

A DLL can theoretically start a separate thread, and that thread can run a message pump. Meanwhile the DLL would be able to return control to the client. We will look at this particular situation later on, but for now we will assume that the client application, regardless of what language it is written in, has created a main window and is running a message pump.

16. Code Sample 3 – A VB6 DLL with Forms

We will use a simple model to demonstrate Form handling. The client will create and show its main window, and then passes all keyboard input (character by character) to a library function called ***vbEcho.***

The ***vbEcho*** library function will echo the keystroke passed in by displaying the keyboard character in a picture box. It will automatically load an display a form containing the picture box the first time it is called. The form will initially be aligned with the client window.

The dll's form will persist until the client application terminates (when the main window is closed). To demonstrate that the form is non-modal, it will also echo keystrokes made when the form itself has the focus. These will be displayed in red, to distinguish them from keystrokes passed by the client.

When the client terminates, any windows created by the DLL are simply destroyed. No VB6 Form events will be signalled. We can, however, use the DLL's ***DllMain*** function to detect client termination and to perform an orderly shutdown. Thus we can unload forms properly, so their Form_Unload event handlers will be called.

There are two special coding conventions we need to use in order to display non-modal forms:

- we can't use the normal VB6 **Show** method, or the application will crash. Instead we should use the **Load** statement to initialise the form, and then call the ***ShowWindow*** API function to make it visible
- if the DLL does manually **Unload** any form, it must take care to keep at least one form (eg. a dummy hidden form) loaded. Once any form is created, the Forms collection cannot subsequently be allowed to become empty, otherwise the application will crash

The precise reasons for these two potentially fatal errors are unclear, but are no doubt related to the fact that we are operating well outside the normal "safe environment" in which VB6 runtime Forms are managed. Fortunately, both cases can easily be avoided.

To make the *CodeSample3* DLL we use the same ***DllMain*** and ***InitVBdll*** code that we used in Code Sample 2. We add two Forms to the project, **BlankForm** and **DllForm. BlankForm** is just an empty form that we will load (but not show) in order to prevent the known problem with the Forms collection.

DllForm is simply a standard form with a PictureBox control, *Picture1*. It should have its **AutoRedraw** property set to True. The code for **DllForm** and for the ***vbEcho*** function is shown below:

```
Dim ClientRect As RECT
Dim FirstCall  As Boolean

Private Sub Form_Load()
   FirstCall = True
   End Sub
Private Sub Form_KeyPress(KeyAscii As Integer)
   If KeyAscii < 256 Then Call EchoChar(KeyAscii, vbRed)
   End Sub
Private Sub Form_Unload(Cancel As Integer)
   MsgBox "Client application has terminated" & vbLf & vbLf _
        & "(DLL Form_Unload)", vbInformation, "CodeSample3.dll"
   End Sub

Sub EchoChar(ByVal Key As Byte, ByVal Colour As Long)
   With Picture1
      .ForeColor = Colour
      If .CurrentX > .Width - 120 Then Picture1.Print ' auto-wrap
      Picture1.Print Chr$(Key);
      End With
   End Sub

Private Sub Form_Resize()
   If WindowState = 1 Then Exit Sub
   If FirstCall Then    ' align window with client
      FirstCall = False
      GetWindowRect ClientWindow, ClientRect
      With ClientRect
         Move Screen.TwipsPerPixelX * .Left,       _
              Screen.TwipsPerPixelY * .Bottom,     _
              Screen.TwipsPerPixelX * .Right - .Left)
         End With
      End If
   Picture1.Move 0, 0, Me.ScaleWidth, Me.ScaleHeight
   End Sub
```

```
Public myForm        As dllForm
Public ClientWindow  As Long

Sub vbEcho(ByVal N As Byte)
   If myForm Is Nothing Then
      ClientWindow = GetForegroundWindow
      Load BlankForm              ' keeps Forms collection non-empty
      Set myForm = New dllForm
      Load myForm
      ShowWindow myForm.hwnd, 1   ' non-modal forms can't use .Show
      End If
   myForm.EchoChar N, 0
   End Sub
```

Figures 12, 13 – CodeSample3 ***DllForm*** *and the* ***vbEcho*** *function*

We also add an extra line to our ***DllMain*** function to detect the client termination event:

```
If lReason = 0 Then Call CloseVBdll
```

This case covers the event where the client application has terminated, and the system is unloading any DLL's it loaded. This allows to perform any necessary "shutdown" tasks. For this example we simply want to unload the form if we have created it. We have placed a MsgBox call in the form's Unload event, so we can confirm that this event has been correctly triggered, even though the client application has terminated. The ***CloseVBdll*** routine is shown here:

```
Sub CloseVBdll()
   If myForm Is Nothing Then Exit Sub
   Unload myForm
   End Sub
```

Figure 14 – CodeSample3 ***CloseVBdll*** *function*

The API declarations needed by the various code segments above is shown below. We can use the normal **Declare** syntax for these, since they will be called only after COM initialisation has been completed.

```
Type RECT
   Left   As Long
   Top    As Long
   Right  As Long
   Bottom As Long
   End Type

Declare Function GetWindowRect Lib "user32" _
   (ByVal hwnd As Long, lpRect As RECT) As Long
Declare Function GetForegroundWindow Lib "user32" () As Long

Declare Function ShowWindow Lib "user32" _
   (ByVal hwnd As Long, ByVal nShow As Long) As Long

Declare Function MoveWindow Lib "user32" _
   (ByVal hwnd As Long, ByVal x As Long, ByVal y As Long, _
    ByVal nWidth As Long, ByVal nHeight As Long, _
    ByVal bRepaint As Long) As Long
```

Figure 15 – CodeSample3 API Declarations

The VBC file commands for this DLL are:

```
Export Module1 vbEcho
Entry  Module1 DllMain
Status
```

Once again, we have assumed here that both functions are in **Module1**

We need a completely new client program to test the *CodeSample3* DLL. Here we present a PowerBasic version, ***Client3***. The listing is shown in three parts. The first part has the program's ***WinMain*** routine, which calls ***OpenMainWindow*** to create the application's main window, then runs a message pump loop:

```
#Compile Exe
#Include "Win32API.inc"

Declare Sub vbEcho  Lib "CodeSample3.dll" Alias "vbEcho" _
   (ByVal C As Long)

Global hModule     As Long    ' appn module handle
Global MainWindow  As Long    ' appn's window handle
Global tLast       As Long    ' last clock update
Global lmsg        As Asciiz * 80
'================================================================
Function WinMain  (ByVal hInstance     As Dword, _
                   ByVal hPrevInstance As Dword, _
                   ByVal lpCmdLine     As Asciiz Ptr, _
                   ByVal iCmdShow      As Long) As Long
   Dim Msg   As    tagMsg
   hModule   = hInstance
   '
   ' create the client application main window
   '
   Call OpenMainWindow
   '
   ' enter the message pump loop
   '
   Do
      If PeekMessage(Msg, 0, 0, 0, 1) Then
         TranslateMessage Msg
         DispatchMessage Msg
         If Msg.Message = %WM_QUIT Then Exit Do
         End If
      Call CheckClock  ' update the time display
      Sleep 0
      Loop
   End Function
'================================================================
Sub CheckClock()
   Dim tNow As Long
   tnow = Timer
   If tNow = tLast Then Exit Sub
   tLast = tNow
   Call PaintWindow
   InvalidateRect MainWindow, ByVal 0, 0
   UpdateWindow MainWindow
   End Sub
```

Figure 16 – ***Client3.bas*** *(Part 1)*

The message pump loop checks for messages, dispatches them if necessary, calls local function ***CheckClock*** , and then surrenders the remainder of the applications CPU timeslice by calling the API function ***Sleep***(0). ***CheckClock*** simply redraws the main window, updating the time display, if the time has changed by 1 second or more since the last update.

The second part of the listing shows the ***OpenMainWindow*** and ***PaintWindow*** routines:

```
Sub OpenMainWindow()
   Dim wce       As WndClassEx
   Dim szAppName As Asciiz * 80

   szAppName         = "CodeSample3_Client"
   wce.cbSize        = SizeOf(wce)
   wce.style         = %CS_HREDRAW Or %CS_VREDRAW
   wce.lpfnWndProc   = CodePtr(WndProc)
   wce.hInstance     = hModule
   wce.hCursor       = LoadCursor(%NULL, ByVal %IDC_ARROW)
   wce.lpszClassName = VarPtr(szAppName)
   wce.hIconSm       = LoadIcon(hModule, ByVal %IDI_APPLICATION)
   RegisterClassEx wce

   MainWindow = CreateWindow(szAppName, _       ' window class name
                    "Client3", _
                    %WS_OVERLAPPEDWINDOW, _     ' window style
                    %CW_USEDEFAULT, _           ' initial x position
                    %CW_USEDEFAULT, _           ' initial y position
                    375, 144, _                 ' initial x size
                    %NULL, _                    ' parent window handle
                    %NULL, _                    ' window menu handle
                    hModule, _                  ' instance handle
                    ByVal 0)                    ' creation parameters
   lmsg = "<< Keystroke Logger >>"
   ShowWindow MainWindow, 1
   End Sub

Sub PaintWindow ()  ' paints window background in "gradient" style
   Dim   hDC        As Long
   Dim   rectFill   As RECT
   Dim   rectClient As RECT
   Dim   fStep      As Single
   Dim   hBrush     As Dword
   Dim   lBand      As Long
   hDC = GetDC(MainWindow)
   GetClientRect WindowFromDC(hDC), rectClient
   fStep = rectClient.nbottom / 200
   For lBand = 0 To 199
      SetRect rectFill, 0, lBand * fStep, rectClient.nright + 1, _
          (lBand + 1) * fStep
      hBrush = CreateSolidBrush(RGB(0, (255-lBand), (255 - lBand)))
      Fillrect hDC, rectFill, hBrush
      DeleteObject hBrush
      Next
   End Sub
```

Figure 17 – ***Client3.bas*** *(Part 2)*

A particular statement of interest here is:

```
   wce.lpfnWndProc   = CodePtr(WndProc)
```

This tells the OS that all messages for this application will be handled by the client's ***WndProc*** function. This is where the client will detect keystrokes and call our DLL's ***vbEcho*** function. The third and final section of this code listing below shows the ***WndProc*** function.

```
Function WndProc (ByVal hWnd As Dword, ByVal wMsg As Dword, _
                  ByVal wParam As Dword, ByVal lParam As Long) _
                  As Long
   '
   ' the message handler for the client app's main window
   '
   Dim hDC    As Dword
   Dim pPaint As PAINTSTRUCT
   Dim tRect  As RECT
   Dim mRect  As RECT
   Dim Cname  As Asciiz * 80
   Dim dOpts  As Long

   Select Case wMsg
      Case %WM_CHAR
         Call vbEcho(wParam)  ' pass keyboard code to the DLL
         SetForegroundWindow MainWindow  ' keeps the focus

'=====================================================================
      Case %WM_PAINT
         hDC = BeginPaint(hWnd, pPaint)
         dOpts = %DT_SINGLELINE Or %DT_CENTER Or %DT_VCENTER
         GetClientRect hWnd, tRect
         SetBkMode hDC, %TRANSPARENT
         SetTextColor hDC, %White
         DrawText hDC, Date$ & "  -   " & Time$, -1, tRect, _
            %DT_SINGLELINE Or %DT_CENTER Or %DT_VCENTER
         trect.nTop = trect.ntop + 54
         SetTextColor hDC, &HC0FFFF
         DrawText hdc, lmsg, -1, tRect, dOpts
         EndPaint hWnd, pPaint
         Function = 1

      Case %WM_ERASEBKGND
         Call PaintWindow
         Function = 1

      Case %WM_DESTROY
         PostQuitMessage 0
         Function = 0

      Case Else  ' pass it to the OS default message handlers
         Function = DefWindowProc(hWnd, wMsg, wParam, lParam)
      End Select
   End Function
```

Figure 18 – ***Client3.bas*** *(Part 3)*

When we run this program, it will send any keystrokes to the DLL. On the first call to the DLL, it creates and displays its form, and echoes each keystroke code passed by the client.

Here we present some snapshots of our test client in action:

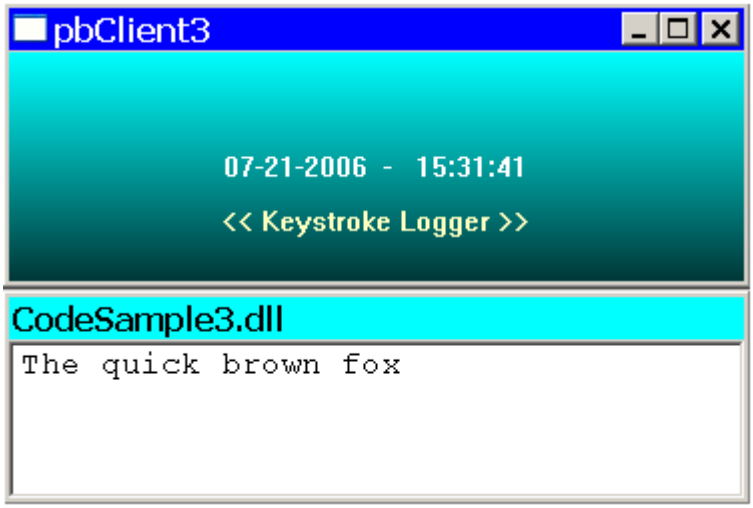

Figure 19 – ***Client3*** *snapshot #1*

For simplicity in this example, we have set the DLL Form's **ControlBox** property to **False.** If you do enable the control box, you need to consider how to manage events such as closing the form. If you wish to keep the form loaded, for example, the **QueryUnload** event handler might be used to simply *hide* the form. In any case, it's important to consider carefully the desired "life cycle" of any forms you show from a DLL (and always keep a "dummy" hidden form object, as discussed at the beginning of this section).

Our DLL is displaying its form non-modally. So we can click on that form, thus giving it the keyboard focus. Now our keystrokes are echoed from the DLL's event handlers, not from client calls:

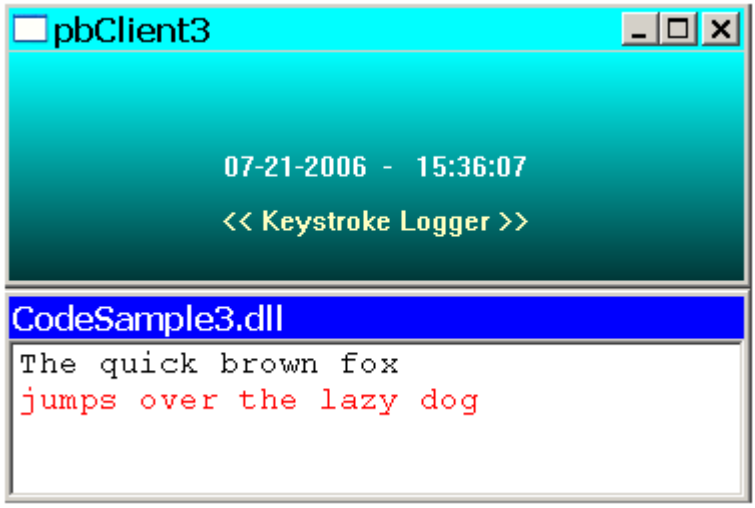

Figure 20 – ***Client3*** *snapshot #2*

17. Miscellaneous Issues for Mixed-Language Interfaces

VB6 and some languages like "C" store strings and some arrays differently, so VB6 library function parameters need careful attention.

Strings

VB6 can perform automatic conversions of string parameters between VB6 and "C" strings (which are used in most system API functions) , but this is only available for a VB6 caller calling a "C" function. The inverse situation needs special handling.

For example, say we added a function to the library of *CodeSample4* called ***vbEchoStr***, which can be passed a "C" string rather than a single character code. A "C" client would declare the parameter as **`char * param[]`**, while a PowerBasic client would declare it as an **`ASCIIZ Ptr.`**

In the VB6 library itself, we declare this parameter as **Byval Long.** The DLL can then make a VB6-compatible copy of the input string - the MVBLC link tool source code includes a function called **CSTRtoVBSTR** that does this.

To returning a "C" string as a function value, we append a null to the string and convert it to an ASCII byte array with the builtin ***StrConv*** function. We then return the *address* of the first byte. The byte array should be a global data item, otherwise the address returned won't be valid. Here is an example of a VB6 function that returns a Timestamp in "C" string format:

```
Public tsBuffer() As Byte   ' global for C string storage

Function vbTimeStamp() As Long
   Dim ts As String
   ts = Format(Now(), "HH:MM:SS  DD MMM YY") & Chr$(0)
   tsBuffer = StrConv(ts, vbFromUnicode)
   vbTimeStamp = VarPtr(tsBuffer(0))
   End Function
```

Figure 21 – A Function returning a "C" String

Arrays

The important thing here is to appreciate the difference between a *dynamic* array and a *fixed* array. VB6's dynamic arrays are referenced indirectly by a descriptor (a SAFEARRAY structure). Fixed arrays are referenced directly by their starting address.

The easiest way to handle array parameters is to require two parameters, one giving the client's array starting address, and the other its length. The library function can then use CopyMemory to access the array elements individually, or perhaps it will be more convenient to copy the entire client array into a dynamic array.

Error Handling

We conclude with a reminder that error handling is crucial to VB6 library management. If we are called by a non-VB client, the VB6 DLL must handle all errors itself. If VB6 signals any error that is not completely handled by the DLL, the client process will crash.

References

[1] Chamberlain, John. *"Take Control of the Compile Process"*, Visual Basic Programmers' Journal, November, 1999. Article available from archives at www.fawcette.com. See John Chamberlain's website for latest article source code and other useful material, johnchamberlain.com

[2] Thé, Lee. *"Inside VB"*, Visual Basic Programmers' Journal, September 1997. Article available from archives at www.fawcette.com

[3] Morris, Peter J. *"Understanding the Visual Basic Compiler (and why it matters)"*. The Mandelbrot Set (International) Limited, 2000. Article available as PDF .

Appendix I: MVBLC Link Control Tool

We list below the complete source code for the Mathimagics Visual Basic Link Control tool. This is a simple, stand-alone VB6 application program that produces an EXE that can be substituted for the VB6 standard linker (NB: make sure that the MVBLC project **Startup Object** is set to ***Sub Main***)

The listing is provided in four separate sections. There are 3 BAS modules (which you can combine into a single one if you wish), and a FRM module for the status display. Before installing this tool, it is essential to make a copy of the standard linker. Go to the VB98 directory in your Visual Studio area, and make a copy of the file **LINK.exe –** call the copy **VBLINK.exe.**

Establish a directory in which you will store the MVBLC project (make it a standard EXE project type), and prepare the source code by pasting from the listings below. Compile the project as LINK.exe in the MVBLC folder, not directly to the VB98 folder, then copy the new LINK.exe to the VB98 folder.

MVBLC will only intervene in the normal linkage procedure if we are making a DLL, and there is a VBC file (a link control command file) present in the same directory as the DLL, and with the same name as the DLL. In all other cases MVBLC operates transparently, it just passes the VB6-specified link command over to the real linker.

A VBC link control file is simply a text file with suffix ".vbc". For example, if you are compiling to *c:\fred\vbtest.dll*, then MVBLC looks for *c:\fred\vbtest.vbc*

Command Syntax

There are five VBC commands: **Export**, **Entry**, **AddDef**, **Status** and **Tidy.** The names themselves are not case-sensitive, but do remember that all LINK symbols ARE case s-ensitive. This means that module names and function names must match exactly with those in the project. Also, note that a *module* is identified by its **Name** property, *not by its file name.*

EXPORT <module> <function names>

Nominates 1 or more functions for export. You can use as many EXPORT commands as you wish. Examples:

```
Export Module1 Function1 Function2
Export Module2 TestFunction
```

AddDEF <alias name> = <function name>

Allows exported functions to be given an alias. Can help solve linking problems for clients that can only call with decorated names (eg. the “C” clients in Appendix II).

ENTRY <module name> <function name>

Nominates a function to be linked as the DLL entrypoint (DllMain) function.

TIDY (no parameters) Tells MVBLC to remove temp files (LIB, DEF, etc) after the DLL has been linked. VB6 itself usually leaves these lying around. They are rarely of much use.

STATUS (no parameters) Tells the link tool to display the DLL’s export table after linking. You can easily see if the new DLL is exporting the correct items.

Error Handling

If you get a name wrong, the real link step might fail. MVBLC pipes the linkage output to a temporary file. If the link does fail you will get two messages – one from the link tool, and also one from the VB6 IDE, with the error message "DLL Load Failed". You just need to correct the errors (most probably in your VBC file) and try the *Make* again.

Module: vbcMain (Main routine)

```
Option Explicit
Option Compare Text
'=========================================================
' mathimagics@yahoo.co.uk
' MVBLC Link Control Tool:   Module "vbcMain"
'=========================================================
'      Jim White,  July 2006
'      Canberra, Australia
'=========================================================
Const VB6FOLDER = "C:\Program Files\MicroSoft Visual Studio\VB98"

Public EXEFILE  As String  ' full pathname of exe/dll file being
Public EXENAME  As String  ' name of exe/dll being built
Dim vbCommand   As String  ' LINK command line passed in by VB6 IDE

Dim Options()   As String  ' Command line tokens
Dim ObjList()   As String  ' list of project OBJ's being linked
Dim EXEPATH     As String  ' Folder containing OBJ files
Dim xList       As String  ' Export request list
Dim F           As Integer ' file unit
Dim ShowStatus  As Boolean ' flag for STATUS command
Dim TidyFlag    As Boolean ' flag for TIDY   command
Dim NormalLink  As Boolean ' did we modify the link in any way?
Dim eMsg        As String  ' link error message

Sub Main()
   NormalLink = True
   vbCommand = Command()    ' make a copy of the command line

   If InStr(vbCommand, "/STATUS:") Then ' internal MVBLC command
      frmLinkInfo.ShowStatus vbCommand
      Exit Sub
      End If
   If InStr(vbCommand, "/ERROR:") Then  ' internal MVBLC command
      frmLinkInfo.ShowError vbCommand
      Exit Sub
      End If

   If InStr(vbCommand, "/DLL") Then LoadVBC  ' check for VBC file

   If NormalLink Then ' no customisations, pass over to std linker
      Execute "VBLINK " & vbCommand, 1
   Else
      RunCustomLink
      End If
   End Sub
```

Module: vbcMain (support routines)

```
Sub RunCustomlink()
   '
   ' Run the real linker as a batch file, so we can check the results
   '
   F = FreeFile
   Open "c:\vbLink.bat" For Output As #F
   Print #F, "cd """ & VB6FOLDER & """"
   Print #F, "VBLINK " & vbCommand & " 1> c:\vbLink.log"
   Print #F, "del c:\vbLink.bat"   ' make the BAT file tidy up
   Close #F
   Execute "c:\vbLink.bat", 1
   If vbLinkError Then Exit Sub
   If TidyFlag Then Call vbTidy
   If ShowStatus Then Call DisplayLinkStatus
   End Sub

Sub vbTidy()
   '
   '  Run a little batch file to tidy up after a VB6 dll build
   '
   F = FreeFile
   Open "c:\vbTidy.bat" For Output As #F
   Print #F, Left$(EXEPATH, 2)  ' assert drive in case it's different
   Print #F, "cd """ & EXEPATH & """"
   Print #F, "if exist " & EXENAME & ".exp del " & EXENAME & ".exp"
   Print #F, "if exist " & EXENAME & ".lib del " & EXENAME & ".lib"
   Print #F, "if exist " & EXENAME & ".def del " & EXENAME & ".def"
   Print #F, "del c:\vbTidy.bat" ' self-deleting bat file
   Close #F
   Execute "c:\vbTidy.bat", 0    ' this doesn't need to be modal
   End Sub

Function vbLinkError() As Boolean
   Dim logentry As String, temp As String
   Dim i As Integer, j As Integer

   If Dir$(EXEFILE) <> "" Then
      If Dir$("c:\vbLink.log") <> "" Then Kill "c:\vbLink.log"
      FixDLL
      Exit Function  ' link was successful
      End If
   vbLinkError = True
   Shell VB6FOLDER & "\Link.exe /ERROR:" & EXEFILE, 1
   End Function

Sub DisplayLinkStatus()
   ' Show export table after a successful custom link
   Shell VB6FOLDER & "\Link.exe /STATUS:" & EXEFILE, 1
   End Sub
```

Module: vbcMain (VBC command handler)

```
Sub LoadVBC()
   Dim xFile   As String       ' link control file (dllname.vbc)
   Dim xName   As String       ' dll export  (.DEF) filename
   Dim xKey    As String       ' control file keyword
   Dim mName   As String       ' Module name
   Dim pName() As String       ' proc names to export
   Dim dName   As String       ' temp for decorated proc name
   Dim EntryFlag As Boolean    ' true if we find an ENTRY command
   Dim j As Long, k As Long

   Options = Split(vbCommand, "/")
   '   Options(0)= the LINK command + link object list
   '             1 = the /ENTRY switch
   '             2 = the /OUT   switch
   '             3 … other switches /BASE, /VERSION, /OPT etc
   '   Fetch EXEpath and EXEname from the /OUT switch.
   '
   For k = 1 To UBound(Options)
      If Left$(Options(k), 4) = "OUT:" Then
         EXEFILE = Mid$(Options(k), 5)
         EXEFILE = Trim$(Replace(EXEFILE, """", ""))
         Exit For
         End If
      Next
   If EXEFILE = "" Then Exit Sub ' unlikely, but ....

   j = InStrRev(EXEFILE, "\")
   EXEPATH = Left$(EXEFILE, j - 1)
   EXENAME = Mid$(EXEFILE, j + 1)
   EXENAME = Left$(EXENAME, Len(EXENAME) - 4)
   '
   ' check for VBC file
   '
   xFile = EXEPATH & "\" & EXENAME & ".vbc"
   If Dir$(xFile) = "" Then
      xFile = CurDir & "\" & EXENAME & ".vbc"
      If Dir$(xFile) = "" Then Exit Sub ' no VBC file, link normally
      End If

   F = FreeFile
   Open xFile For Input As #F
   Do Until EOF(F)
      Line Input #F, xKey  ' comments are supported (use “;” or “'”)
      j = InStr(xKey, ";"): If j Then xKey = Left$(xKey, j - 1)
      j = InStr(xKey, "'"): If j Then xKey = Left$(xKey, j - 1)
      xKey = Trim(xKey)
      While InStr(xKey, Space(2))
         xKey = Replace(xKey, Space(2), Space(1))
         Wend
      pName = Split(xKey, " ")

      Select Case pName(0)

         Case "Status":    ShowStatus = True
         Case "Tidy":      TidyFlag = True
```

Module: vbcMain (VBC command handler - ctd)

```
         Case "Export"          ' EXPORT <module> <proclist>
            If UBound(pName) > 1 Then
               mName = pName(1) ' module name
               For j = 2 To UBound(pName)
                  dName = "?" & pName(j) & "@" & mName & "@@AAGXXZ"
                  xList = xList & "," & pName(j) & " = " & dName
                  Next
               End If
         Case "Entry"           ' ENTRY <module> <procname>
            If UBound(pName) = 2 Then
               mName = pName(1) ' module name
               dName = "?" & pName(2) & "@" & mName & "@@AAGXXZ"
               xList = xList & "," & pName(2) & " = " & dName
               For k = 1 To UBound(Options)
                  If Left$(Options(k), 6) = "ENTRY:" Then
                     Options(k) = "ENTRY:" & pName(2) & " "
                     vbCommand = " " & Join(Options, "/")
                     EntryFlag = True
                     Exit For
                     End If
                  Next
               End If
         Case "AddDef"          ' AddDEF <aliasname> = <name>
            If pName(2) = "=" Then  ' we neeed the decorated <name>
               dName = pName(3)     ' if available
               j = InStr(xList, "," & pName(3) & " = ")
               If j Then                    '   if it is in our list
                  dName = Mid(xList, j + Len(pName(3)) + 4)
                  k = InStr(dName, ",")
                  If k Then dName = Left(dName, k - 1)
                  End If
               xList = xList & "," & pName(1) & " = " & dName
               End If
         End Select
NextLine:
      Loop
   Close #F
   If xList = "" Then Exit Sub
   ' Custom link: build DEF file, add /DEF switch to the command line
   NormalLink = False
   If EntryFlag Then xList = xList & ",__vbaS"  ' export __vbaS
   pName = Split(xList, ",")
   xName = EXEPATH & "\" & EXENAME & ".def"
   F = FreeFile
   Open xName For Output As #F
   Print #F, "LIBRARY "; EXENAME
   Print #F, "EXPORTS"
   For j = 1 To UBound(pName)
      Print #F, Space(3) & pName(j)
      Next
   Close #F
   vbCommand = vbCommand & " /DEF:""" & xName & """"
   End Sub
```

Module: vbcCommand (“Execute” routine)

This module has a generic routine for executing any command line modally.

```
Type STARTUPINFO  ' structure used with CreateProcess API
   cb              As Long
   lpReserved      As String
   lpDesktop       As Long
   lpTitle         As String
   dwX             As Long
   dwY             As Long
   dwXSize         As Long
   dwYSize         As Long
   dwXCountChars   As Long
   dwYCountChars   As Long
   dwFillAttribute As Long
   dwFlags         As Long
   wShowWindow     As Integer
   cbReserved2     As Integer
   lpReserved2     As Long
   hStdInput       As Long
   hStdOutput      As Long
   hStdError       As Long
   End Type

Type PROCESS_INFORMATION
   hProcess        As Long
   hThread         As Long
   dwProcessID     As Long
   dwThreadID      As Long
   End Type

Declare Function WaitForSingleObject Lib "kernel32" (ByVal _
         hHandle As Long, ByVal dwMilliseconds As Long) As Long

Declare Function CreateProcessA Lib "kernel32" (ByVal _
  lpAppName As Long, ByVal lpCommandLine As String, _
  ByVal lpProcessAtts As Long, ByVal lpThreadAtts As Long, _
  ByVal bInheritHandles As Long, ByVal dwCreationFlags As Long, _
  ByVal lpEnvironment As Long, ByVal lpCurrentDirectory As Long, _
  lpStartupInfo As STARTUPINFO, lpProcessInformation As _
  PROCESS_INFORMATION) As Long

Sub Execute(WinCommand As String, ByVal Modal As Long)
   Const NORMAL_PRIORITY_CLASS = &H20&
   Dim ProcInfo   As PROCESS_INFORMATION
   Dim StartInfo  As STARTUPINFO
   StartInfo.cb = Len(StartInfo)
   StartInfo.dwFlags = 1

   Call CreateProcessA(0&, WinCommand, 0&, 0&, 1&, _
            NORMAL_PRIORITY_CLASS, 0&, 0&, _
            StartInfo, ProcInfo)
   If Modal Then Call WaitForSingleObject(ProcInfo.hProcess, -1)
   End Sub
```

Module: vbcTools (Utility functions)

This module has some utility functions for inspecting the DLL's export table and producing the status report.

```
Option Explicit
Option Compare Text


Type LOADED_IMAGE
   ModuleName       As Long
   hFile            As Long
   MappedAddress    As Long      ' Base address of mapped file
   pFileHeader      As Long      ' Pointer to IMAGE_PE_FILE_HEADER
   pLastRvaSection  As Long
   NumberOfSections As Long
   pSections        As Long      ' Pointer to first COFF section header
   Characteristics  As Long      ' Image characteristics value
   fSystemImage     As Byte
   fDOSImage        As Byte
   FLink            As Long
   Blink            As Long
   SizeOfImage      As Long
   End Type

Type IMAGE_DATA_DIRECTORY
   RVA   As Long
   size  As Long
   End Type

Declare Function MapAndLoad Lib "Imagehlp.dll" ( _
   ByVal ImageName As String,   ByVal DLLPath As String, _
   LoadedImage As LOADED_IMAGE, DotDLL As Long, _
   ReadOnly As Long) As Long

Declare Function UnMapAndLoad Lib "Imagehlp.dll" ( _
   LoadedImage As LOADED_IMAGE) As Long

Declare Function ImageRvaToVa Lib "Imagehlp.dll" ( _
   ByVal NTHeaders As Long, ByVal Base As Long, _
   ByVal RVA As Long, ByVal LastRvaSection As Long) As Long

Declare Sub CopyMemory Lib "kernel32" Alias "RtlMoveMemory" ( _
   lpvDest As Any, lpvSource As Any, ByVal cbCopy As Long)

Declare Sub Sleep Lib "kernel32" (ByVal nMilliseconds As Long)

Declare Function lstrlenA Lib "kernel32" (ByVal lpsz As Long) As Long
```

Module: vbcTools (ctd)

```
Type IMAGE_OPTIONAL_HEADER
   Magic                    As Integer
   MajorLinkerVersion       As Byte
   MinorLinkerVersion       As Byte
   SizeOfCode               As Long
   SizeOfInitializedData    As Long
   SizeOfUninitializedData  As Long
   AddressOfEntryPoint      As Long
   BaseOfCode               As Long
   BaseOfData               As Long
   ImageBase                As Long
   SectionAlignment         As Long
   FileAlignment            As Long
   MajorOperatingSystemVersion  As Integer
   MinorOperatingSystemVersion  As Integer
   MajorImageVersion        As Integer
   MinorImageVersion        As Integer
   MajorSubsystemVersion    As Integer
   MinorSubsystemVersion    As Integer
   Win32VersionValue        As Long
   SizeOfImage              As Long
   SizeOfHeaders            As Long
   CheckSum                 As Long
   Subsystem                As Integer
   DllCharacteristics       As Integer
   SizeOfStackReserve       As Long
   SizeOfStackCommit        As Long
   SizeOfHeapReserve        As Long
   SizeOfHeapCommit         As Long
   LoaderFlags              As Long
   NumberOfRvaAndSizes      As Long
   DataDirectory(0 To 15)   As IMAGE_DATA_DIRECTORY
   End Type

Type IMAGE_COFF_HEADER
   Machine                  As Integer
   NumberOfSections         As Integer
   TimeDateStamp            As Long
   PointerToSymbolTable     As Long
   NumberOfSymbols          As Long
   SizeOfOptionalHeader     As Integer
   Characteristics          As Integer
   End Type

Type IMAGE_PE_FILE_HEADER
   Signature                As Long
   FileHeader               As IMAGE_COFF_HEADER
   OptionalHeader           As IMAGE_OPTIONAL_HEADER
   End Type
```

Module: vbcTools (ctd)

```
Type IMAGE_EXPORT_DIRECTORY_TABLE
   Characteristics            As Long
   TimeDateStamp              As Long
   MajorVersion               As Integer
   MinorVersion               As Integer
   Name                       As Long
   Base                       As Long
   NumberOfFunctions          As Long
   NumberOfNames              As Long
   pAddressOfFunctions        As Long
   ExportNamePointerTableRVA  As Long
   pAddressOfNameOrdinals     As Long
   End Type

Public LoadImage  As LOADED_IMAGE
Dim peheader   As IMAGE_PE_FILE_HEADER
Dim exportdir  As IMAGE_EXPORT_DIRECTORY_TABLE
Dim vaEntryPoint   As Long
Dim rvaEntryPoint  As Long
Dim dllBaseAddress As Long
Dim procTable      As Long
Dim procAddress    As Long
Dim ExportNamePointerTableVA As Long
Dim ImportNamePointerTableVA As Long
Dim rvaImportDirTable As Long
Dim rvaExportDirTable As Long
Dim vaImportDirTable  As Long
Dim vaExportDirTable  As Long

' The following string must match (in uppercase), the fixed
' DLL name used in the type library (vbLibraryHelper.tlb)

Const FixDLLname = "VBLIBRARYHELPER_MATHIMAGICS.DLL"

Sub LoadDLL()
   If MapAndLoad(EXEFILE, "", LoadImage, 1, 1) = 0 Then Exit Sub
   CopyMemory peheader, ByVal LoadImage.pFileHeader, 256
   rvaEntryPoint = peheader.OptionalHeader.AddressOfEntryPoint
   dllBaseAddress = peheader.OptionalHeader.ImageBase
   If rvaEntryPoint Then
      vaEntryPoint = ImageRvaToVa(LoadImage.pFileHeader, _
           LoadImage.MappedAddress, rvaEntryPoint, 0&)
      End If

   rvaExportDirTable = peheader.OptionalHeader.DataDirectory(0).RVA
   If rvaExportDirTable Then
      vaExportDirTable = ImageRvaToVa(LoadImage.pFileHeader, _
           LoadImage.MappedAddress, rvaExportDirTable, 0&)
      End If

   rvaImportDirTable = peheader.OptionalHeader.DataDirectory(1).RVA
   If rvaImportDirTable Then
      vaImportDirTable = ImageRvaToVa(LoadImage.pFileHeader, _
           LoadImage.MappedAddress, rvaImportDirTable, 0&)
      End If
   End Sub
```

Module: vbcTools (ctd)

```
Sub FixDLL()
   '
   ' If the TLB is used, this DLL needs self-referencing. If there
   '   is an export table entry matches the TLB fixed name, change
   '   it to be the name of this DLL.
   '
   Dim buf() As Byte, oldkey() As Byte, keylen As Integer
   Dim newkey() As Byte
   Dim i&, j&, k&, b&, F%, fsize&

   LoadDLL                    ' load the DLL image (this will get
   UnMapAndLoad LoadImage     ' the offset of its Import Table)

   keylen = Len(FixDLLname)
   oldkey = StrConv(FixDLLname & Chr$(0), vbFromUnicode)
   newkey = StrConv(UCase(EXENAME & ".dll") & Chr$(0), vbFromUnicode)
   F = FreeFile
   Open EXEFILE For Binary As #F
   fsize = LOF(F)
   ReDim buf(fsize - 1)
   Get #F, , buf

   Dim pImportTable   As Long
   Dim LookupTableRVA As Long
   Dim DLLNameRVA     As Long
   Dim DLLname        As String

   pImportTable = rvaImportDirTable  ' set by LoadDLL
   Do
      CopyMemory LookupTableRVA, buf(pImportTable), 4
      CopyMemory DLLNameRVA, buf(pImportTable + 12), 4
      If LookupTableRVA = 0 And DLLNameRVA = 0 Then Exit Do
      DLLname = CSTRtoVBSTR(VarPtr(buf(DLLNameRVA)))
      If DLLname = FixDLLname Then
         Seek F, DLLNameRVA + 1
         Put F, , newkey  ' Fix the DLL Import Table entry
         Exit Do
         End If
      pImportTable = pImportTable + 20
      Loop
   Close #F
   End Sub

Function CSTRtoVBSTR(ByVal lpsz As Long) As String
   Dim i As Long, cChars As Long    ' C-to-VB string converter
   cChars = lstrlenA(lpsz)
   CSTRtoVBSTR = String$(cChars, 0)
   CopyMemory ByVal StrPtr(CSTRtoVBSTR), ByVal lpsz, cChars
   CSTRtoVBSTR = StrConv(CSTRtoVBSTR, vbUnicode)
   i = InStr(CSTRtoVBSTR, Chr$(0))
   If i > 0 Then CSTRtoVBSTR = Left$(CSTRtoVBSTR, i - 1)
   End Function
```

Module: vbcTools (ctd)

```
Function GetExports() As String
  Dim i      As Long,    nNames As Long
  Dim sName  As String,  epName As String
  Dim pNext  As Long,    lNext  As Long
  Dim epFlag As Boolean, xpFlag As Boolean, nxp As Integer
  Dim xList  As String,  iTag   As String

  xList = GetImports  ' get list of self-Imported names, if any

  If vaExportDirTable = 0 Then
     GetExports = GetExports & vbLf & "  no access to Export Table)"
     Exit Function
     End If

  CopyMemory exportdir, ByVal vaExportDirTable, LenB(exportdir)
  procTable = ImageRvaToVa(LoadImage.pFileHeader, _
    LoadImage.MappedAddress, exportdir.pAddressOfFunctions, 0)
  nNames = exportdir.NumberOfNames
  If nNames = 0 Then Exit Function

  ExportNamePointerTableVA = ImageRvaToVa(LoadImage.pFileHeader, _
    LoadImage.MappedAddress, _
    exportdir.ExportNamePointerTableRVA, 0&)

  pNext = ExportNamePointerTableVA
  CopyMemory lNext, ByVal pNext, 4
  For i = 0 To nNames - 1
     lNext = ImageRvaToVa(LoadImage.pFileHeader, _
          LoadImage.MappedAddress, lNext, 0&)
     sName = CSTRtoVBSTR(lNext)
     CopyMemory procAddress, ByVal procTable, 4
     epFlag = (procAddress = rvaEntryPoint) 'is this the entrypoint?
     xpFlag = InStr(xList, vbLf & sName)
     iTag = Space(4)
     If epFlag Then iTag = Space(2) & "* ": epName = sName
     If xpFlag Then iTag = Space(2) & "~ ": nxp = nxp + 1
     iTag = iTag & Hex(procAddress + dllBaseAddress) & ": " & sName
     GetExports = GetExports & vbLf & iTag
     pNext = pNext + 4
     procTable = procTable + 4
     CopyMemory lNext, ByVal pNext, 4
     Next

  If Len(epName) Or nxp Then   ' add self-import info
     GetExports = GetExports & vbLf & "  -------------------"
     If nxp Then GetExports = GetExports & vbLf & " ~ = auto-import"
     If Len(epName) Then _
          GetExports = GetExports & vbLf & " * = entrypoint"
  Else
     GetExports = GetExports & vbLf & Space(4) & _
          Hex(rvaEntryPoint + dllBaseAddress) & ":  <entrypoint>"
     End If
  GetExports = GetExports & vbLf & "  -------------------"
  End Function
```

Module: vbcTools (ctd)

```
Function GetImports() As String
   Dim i      As Long,    nNames As Long
   Dim sName  As String,  pNames As String
   Dim pNext  As Long,    lNext  As Long

   If vaImportDirTable = 0 Then Exit Function

   Dim pImportTable   As Long
   Dim pLookupTable   As Long
   Dim pLookupEntry   As Long
   Dim LookupTableRVA As Long
   Dim DLLNameRVA     As Long
   Dim DLLname        As String

   pImportTable = vaImportDirTable  ' set by LoadDLL

   Do
      CopyMemory LookupTableRVA, ByVal pImportTable, 4
      CopyMemory DLLNameRVA, ByVal pImportTable + 12, 4
      If LookupTableRVA = 0 And DLLNameRVA = 0 Then Exit Do
      pLookupTable = ImageRvaToVa(LoadImage.pFileHeader, _
           LoadImage.MappedAddress, LookupTableRVA, 0&)
      DLLNameRVA = ImageRvaToVa(LoadImage.pFileHeader,  _
           LoadImage.MappedAddress, DLLNameRVA, 0&)
      DLLname = CSTRtoVBSTR(DLLNameRVA)
      If DLLname = EXENAME Then
         GoSub GetProcList
         Exit Function
         End If
      pImportTable = pImportTable + 20
      Loop
   Exit Function

GetProcList:
   Do While pLookupTable
      CopyMemory pLookupEntry, ByVal pLookupTable, 4
      If pLookupEntry = 0 Then Exit Do
      pNext = ImageRvaToVa(LoadImage.pFileHeader, _
      LoadImage.MappedAddress, pLookupEntry, 0&)
      sName = CSTRtoVBSTR(pNext + 2)
      pNames = pNames & vbLf & sName
      nNames = nNames + 1
      pLookupTable = pLookupTable + 4
      Loop
   GetImports = pNames
   Return
   End Function
```

Module: DllForm

We show only the code here. Create the form, naming it **DllForm,** and set its **BorderStyle** to 5. Insert a listbox, **List1.** Set the listbox font to Courier New (or some fixed font).

```
Private Sub Form_Resize()
   List1.Move 0, 0, Me.ScaleWidth, Me.ScaleHeight
   End Sub

Sub ShowStatus(vbCommand As String)
   Dim j%, token$()
   j = InStr(vbCommand, "/STATUS:"):  EXEFILE = Mid(vbCommand, j + 8)
   j = InStrRev(EXEFILE, "\"):        EXENAME = Mid$(EXEFILE, j + 1)
   Show
   LoadDLL
   With frmLinkInfo.List1
      .AddItem ""
      .AddItem "  Export List: " & EXENAME
      token = Split(GetExports(), vbLf)
      For j = 0 To UBound(token): .AddItem token(j): Next
      .AddItem Format(Now, "  HH:MM:SS  DD MMM YY")
      .ListIndex = .ListCount - 1: DoEvents
      .ListIndex = -1
      End With
   UnMapAndLoad LoadImage
   End Sub

Sub ShowError(vbCommand As String)
   Dim F%, j%, temp$, fLine$
   j = InStr(vbCommand, "/STATUS:"): EXEFILE = Mid(vbCommand, j + 8)
   j = InStrRev(EXEFILE, "\"):       EXENAME = Mid$(EXEFILE, j + 1)
   Show
   List1.AddItem ""
   List1.AddItem "An unexpected link error has occurred"
   List1.AddItem EXENAME & " link failed"
   List1.AddItem ""
   F = FreeFile
   On Error GoTo BadSign
   Open "c:\vbLink.log" For Input As #F
   Do
      Line Input #F, fLine
      j = InStr(fLine, "error")
      If j Then
         fLine = Mid$(fLine, j)
         j = InStr(fLine, """")
         If j Then
            temp = Left$(fLine, j - 1)
            j = InStr(fLine, """ (")
            If j Then fLine = temp & Mid$(fLine, j + 2)
            End If
         List1.AddItem "> " & fLine
         End If
      Loop Until EOF(F)
   Close #F
   Exit Sub
BadSign:
   List1.AddItem "The log file is not available"
   End Sub
```

Appendix II: "C" Versions of the Client Test Programs

Here we provide "C" versions of the test-client programs ***Client1***, ***Client2*** and ***Client3*** are listed below. These have been tested with the **gcc** compiler distributed with MinGW. The **gcc** linker produces an EXE in which the external API names are decorated with "*@n*", where *n* is the number of bytes in the argument list (ie. 4 * the number of parameters). For example, **Client1** looks for ***IsPrime@4*** and ***DllGetClassObject$12*** in *CodeSample1.dll*. This problem is resolved by linking *CodeSample1.dll* with the appropriate alias definitions added to the VBC file. These commands are given below where appropriate.

Client1.c

```
#include <windows.h>
#include <stdio.h>

int WINAPI IsPrime(int x);
IID rIID;

void RegisterThisApp(HINSTANCE hModule) {
   WNDCLASSEX      wce;

   char szAppName[]  = "CodeSample1_Client";
   wce.cbSize        = sizeof(WNDCLASSEX);
   wce.style         = CS_HREDRAW | CS_VREDRAW;
   wce.cbClsExtra    = 0;
   wce.cbWndExtra    = 0;
   wce.hInstance     = hModule;
   wce.hCursor       = LoadCursor(NULL, IDC_ARROW);
   wce.hIcon         = NULL;
   wce.hbrBackground = NULL;
   wce.lpszMenuName  = NULL;
   wce.lpszClassName = szAppName;
   wce.lpfnWndProc   = DefWindowProc;
   wce.hIconSm       = LoadIcon(hModule, IDI_APPLICATION);
   RegisterClassEx (&wce);
   }
void test(int n) {
   if (IsPrime(n)) printf(" %d is prime\n",n);
               else printf(" %d is not prime\n",n);
   }

int WINAPI WinMain (HINSTANCE hModule, HINSTANCE hPrevInstance,
                       LPSTR command_line, int nCmdShow)  {
   char option[2];
   int dummy;
   RegisterThisApp(hModule);
   printf("Call COM initialiser? ");   gets(option);
   if ( (strcmp(option, "y") == 0) || (strcmp(option, "Y") == 0)) {
      rIID.Data1 = 1; rIID.Data4[0] = 0xc0; rIID.Data4[7] = 0x46;
      DllGetClassObject((REFCLSID) &dummy, &rIID, (PVOID*) &dummy);
      }
   test(41);
   test(42);
   test(-43);
   }
```

Additional VBC Commands

The decorated names generated by **gcc** require that *CodeSample1.dll* be built with aliases. These should be added to the end of the standard VBC file described above:

- `AddDef IsPrime@4 = IsPrime`
- `AddDef DllGetClassObject@12 = DllGetClassObject`

Compile and Link Commands

- `C:\MinGW\bin\gcc -c Client1.c  -O3`
- `c:\MinGw\bin\gcc -o Client1 Client1.o CodeSample1.dll -lgdi32 –lm`

Differences between GCC and PowerBasic versions

The input and output is done via **stdin** and **stdout.** This means that the only message box displayed will be the error message from the DLL. Also, we find that the EXE's produced by **gcc** won't operate correctly unless they begin by registering a window class. The reasons are unclear at this stage, but without the ***RegisterThisApp*** call the **gcc**-linked **Client1** program crashes when it calls the DLL.

Client2.c

```
#include <windows.h>
#include <stdio.h>
int WINAPI IsPrime(int x);

void RegisterThisApp(HINSTANCE hModule) {
   WNDCLASSEX      wce;
   char szAppName[]  = "CodeSample1_Client";
   wce.cbSize        = sizeof(WNDCLASSEX);
   wce.style         = CS_HREDRAW | CS_VREDRAW;
   wce.cbClsExtra    = 0;
   wce.cbWndExtra    = 0;
   wce.hInstance     = hModule;
   wce.hCursor       = LoadCursor(NULL, IDC_ARROW);
   wce.hIcon         = NULL;
   wce.hbrBackground = NULL;
   wce.lpszMenuName  = NULL;
   wce.lpszClassName = szAppName;
   wce.lpfnWndProc   = DefWindowProc;
   wce.hIconSm       = LoadIcon(hModule, IDI_APPLICATION);
   RegisterClassEx (&wce);
   }
void test(int n) {
   if (IsPrime(n)) printf(" %d is prime\n",n);
               else printf(" %d is not prime\n",n);
   }
int WINAPI WinMain (HINSTANCE hModule, HINSTANCE hPrevInstance,
                      LPSTR command_line, int nCmdShow)  {
   RegisterThisApp(hModule);
   test(41);
   test(42);
   test(-43);
   }
```

Additional VBC Commands

Add the following line to *CodeSample2.vbc* when making *CodeSample2.dll:*

- `AddDef IsPrime@4 = IsPrime`

Compile and Link Commands

- `C:\MinGW\bin\gcc -c Client2.c  -O3`
- `c:\MinGw\bin\gcc -o Client2 Client2.o CodeSample2.dll -lgdi32 –lm`

Client3.c (part 1 of 3)

```
#include "windows.h"
#include <time.h>

void WINAPI vbEcho(int);

HINSTANCE hModule;     // appn module handle
HWND      MainWindow;  // appn's window handle
time_t    tLast;       // last clock update
void      CheckClock();
void      CreateMainWindow();
void      PaintWindow();
LRESULT CALLBACK
          WndProc (HWND hWnd, UINT wMsg,
                   WPARAM wParam, LPARAM lParam);

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                      LPSTR command_line, int nCmdShow)  {
   MSG         msg;
   hModule = hInstance;
   CreateMainWindow();
   do {                    // the message pump
      if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) {
         TranslateMessage(&msg);
         DispatchMessage(&msg);
         if (msg.message == WM_QUIT) break;
         }
      CheckClock();
      Sleep(0);
      } while(1);
   }
void CheckClock() {
   time_t tNow;
   time (&tNow);
   if (tNow != tLast) {
      tLast = tNow;
      PaintWindow();
      InvalidateRect (MainWindow, 0, 0);
      UpdateWindow   (MainWindow);
      }
   }
```

Client3.c (part 2 of 3)

```
void CreateMainWindow() {
   WNDCLASSEX      wce;
   char szAppName[]  = "CodeSample3_Client";
   wce.cbSize        = sizeof(WNDCLASSEX);
   wce.style         = CS_HREDRAW | CS_VREDRAW;
   wce.lpfnWndProc   = WndProc;
   wce.cbClsExtra    = 0;
   wce.cbWndExtra    = 0;
   wce.hInstance     = hModule;
   wce.hCursor       = LoadCursor(NULL, IDC_ARROW);
   wce.hIcon         = NULL;
   wce.hbrBackground = NULL;
   wce.lpszMenuName  = NULL;
   wce.lpszClassName = szAppName;
   wce.hIconSm       = LoadIcon(hModule, IDI_APPLICATION);

   RegisterClassEx (&wce);

   MainWindow = CreateWindow(szAppName,       // window class name
                        "Mathimagics Demo 1: Client Application",
                        WS_OVERLAPPEDWINDOW,  // window style
                        CW_USEDEFAULT,        // initial x position
                        CW_USEDEFAULT,        // initial y position
                        375, 200,             // initial x size
                        NULL,                 // parent window handle
                        NULL,                 // window menu handle
                        hModule,              // instance handle
                        NULL);

   ShowWindow (MainWindow, 1);
   }

void PaintWindow () {
    HDC    hDC;
    RECT   rectFill;
    RECT   rectClient;
    int    fStep1, fStep2;
    HBRUSH hBrush;
    int    lOnBand;
    hDC =  GetDC(MainWindow);

    GetClientRect (WindowFromDC(hDC), &rectClient);

    for (lOnBand = 0; lOnBand < 200; lOnBand++) {
        fStep1 = lOnBand * rectClient.bottom / 200;
        fStep2 = (lOnBand+1) * rectClient.bottom / 200;
        SetRect (&rectFill, 0, fStep1, rectClient.right + 1, fStep2);
        hBrush = CreateSolidBrush(RGB(0, 0, (255 - lOnBand)));
        FillRect (hDC, &rectFill, hBrush);
        DeleteObject(hBrush);
        }
    }
```

Client3.c (part 3 of 3)

```
LRESULT CALLBACK WndProc (HWND hWnd, UINT wMsg,
                          WPARAM wParam, LPARAM lParam)
  {
  HDC          hDC;
  PAINTSTRUCT  pPaint;
  RECT         tRect;
  DWORD        dOpts;

  switch (wMsg) {
     case WM_PAINT:
        hDC = BeginPaint(hWnd, &pPaint);
        dOpts = DT_SINGLELINE | DT_CENTER | DT_VCENTER;
        GetClientRect(hWnd, &tRect);
        SetBkMode (hDC, TRANSPARENT);
        SetTextColor (hDC, RGB(255,255,255));
        DrawText (hDC, ctime(&tLast), 24, &tRect, dOpts);
        tRect.top += 54;
        SetTextColor (hDC, RGB(255,255,192));
        DrawText (hDC, "<< Keystroke Logger (\"C\" version) >>",
            -1, &tRect, dOpts);
        EndPaint (hWnd, &pPaint);
        return 1;

     case WM_ERASEBKGND:
        hDC = (HDC) wParam;
        PaintWindow();
        return 1;

     case WM_DESTROY:
        PostQuitMessage (0);
        return 0;

     case WM_CHAR:
        vbEcho(wParam);         //  ***  DLL  ***
        SetForegroundWindow(MainWindow);
        return 0;
        }
  return DefWindowProc(hWnd, wMsg, wParam, lParam);
  }
```

Additional VBC Commands

Add the following line to *CodeSample3.vbc* when making *CodeSample3.dll:*

- `AddDef vbEcho@4 = vbEcho`

Compile and Link Commands

- `C:\MinGW\bin\gcc -c Client3.c  -O3`
- `c:\MinGw\bin\gcc -o Client3 Client3.o CodeSample3.dll`
  `-lgdi32 –lm`